

Отчёт по лабораторной работе №13

**Средства, применяемые при разработке программного обеспечения в
ОС типа UNIX/Linux**

Щербак Маргарита Романовна

2022

1 Цель работы:

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

2 Теоретическое введение:

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gсс, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. Файлы с расширением (суффиксом) .с воспринимаются gсс как программы на языке C, файлы с расширением .сс или .C — как файлы на языке C++, а файлы с расширением .о считаются объектными.

Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами. Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса.

В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды — собственно действия, которые необходимо выполнить для достижения цели.

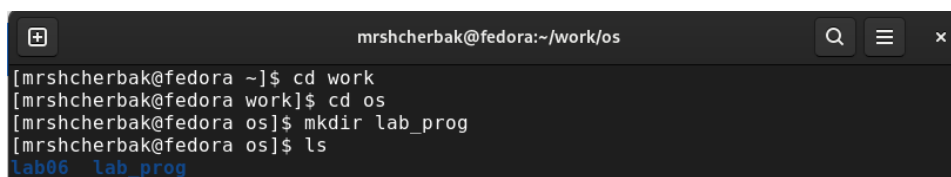
Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения

ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле.

Ещё одним средством проверки исходных кодов программ, написанных на языке C, является утилита splint. Эта утилита анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки. В отличие от компилятора C анализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.

3 Выполнение лабораторной работы:

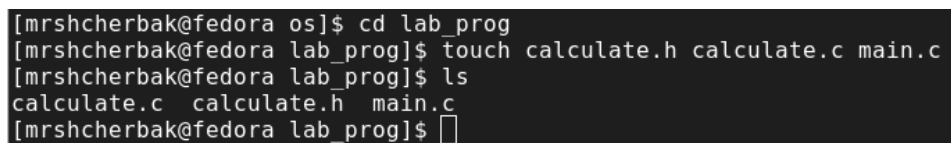
1. В домашнем каталоге создала подкаталог ~/work/os/lab_prog.(Рис. 3.1).



```
mrshcherbak@fedora:~/work/os
[mrshcherbak@fedora ~]$ cd work
[mrshcherbak@fedora work]$ cd os
[mrshcherbak@fedora os]$ mkdir lab_prog
[mrshcherbak@fedora os]$ ls
lab06  lab_prog
```

Рис. 3.1: Создание каталога lab_prog

2. Создала в нём файлы: calculate.h, calculate.c, main.c. (Рис. 3.2).

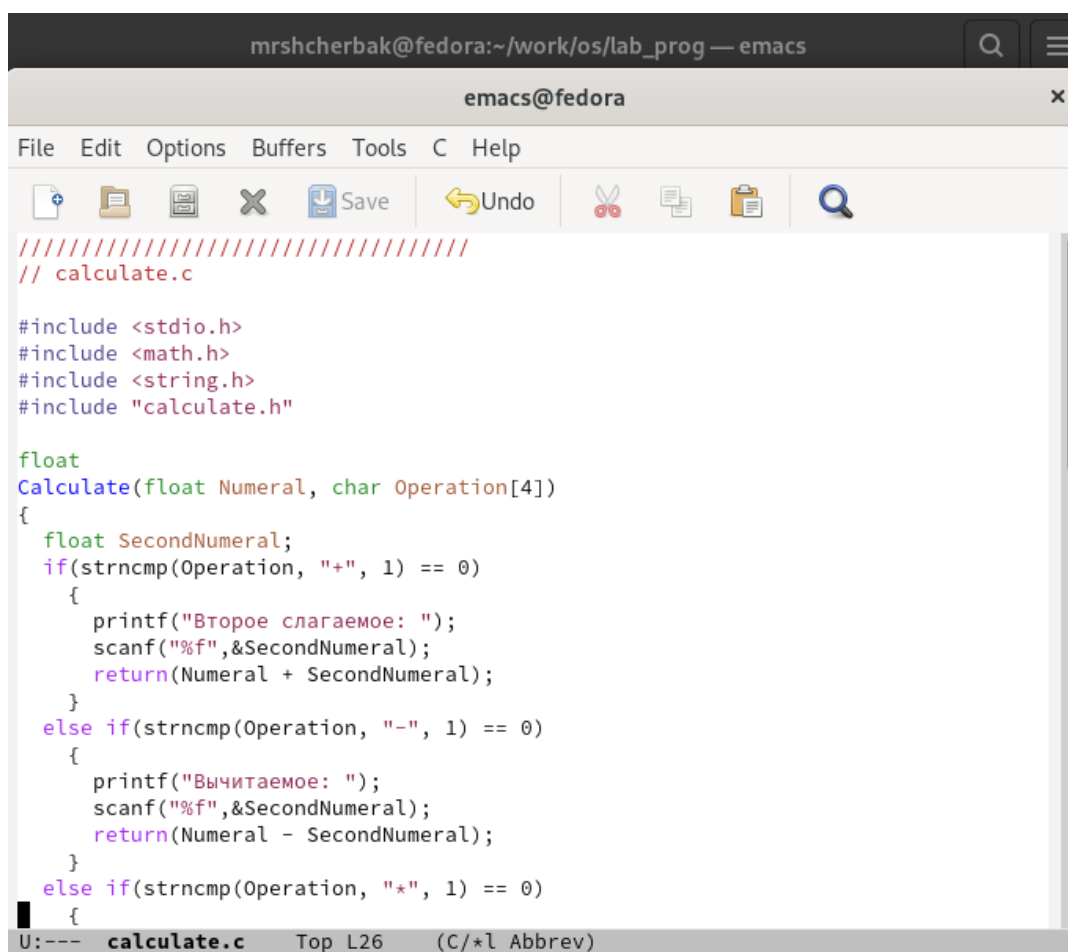


```
[mrshcherbak@fedora os]$ cd lab_prog
[mrshcherbak@fedora lab_prog]$ touch calculate.h calculate.c main.c
[mrshcherbak@fedora lab_prog]$ ls
calculate.c calculate.h main.c
[mrshcherbak@fedora lab_prog]$
```

Рис. 3.2: Создание файлов

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять \sin , \cos , \tan . При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.

3. Открыла редактор emacs и записала реализацию функций калькулятора в файле calculate.c (Содержимое данного файла в полном объеме представлено в методичке к этой лабораторной работе): (Рис. 3.3).



The screenshot shows the Emacs editor interface. The title bar at the top reads 'mrshcherbak@fedora:~/work/os/lab_prog — emacs'. Below it, the window title is 'emacs@fedora'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The main text area displays the following C code for 'calculate.c':

```
////////////////////////////////////
// calculate.c

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

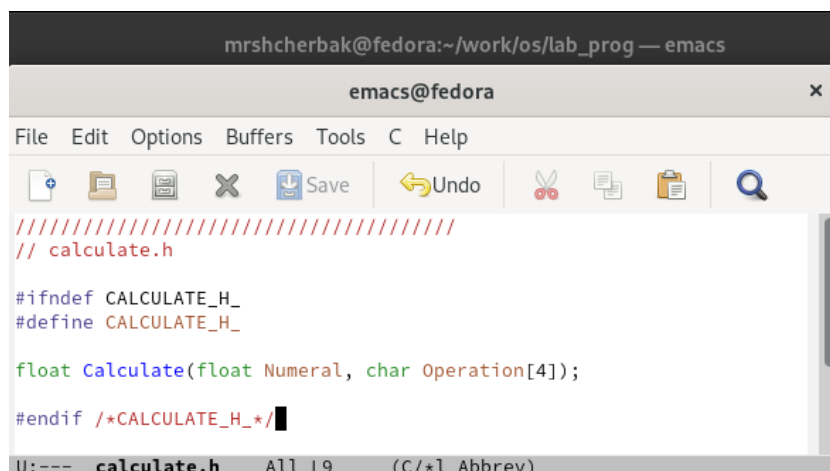
float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strncmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strncmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strncmp(Operation, "*", 1) == 0)
    {

```

The status bar at the bottom shows 'U:--- calculate.c Top L26 (C/*l Abbrev)'.

Рис. 3.3: Реализация функций калькулятора

4. В этом же редакторе открыла интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора: (Рис. 3.4).

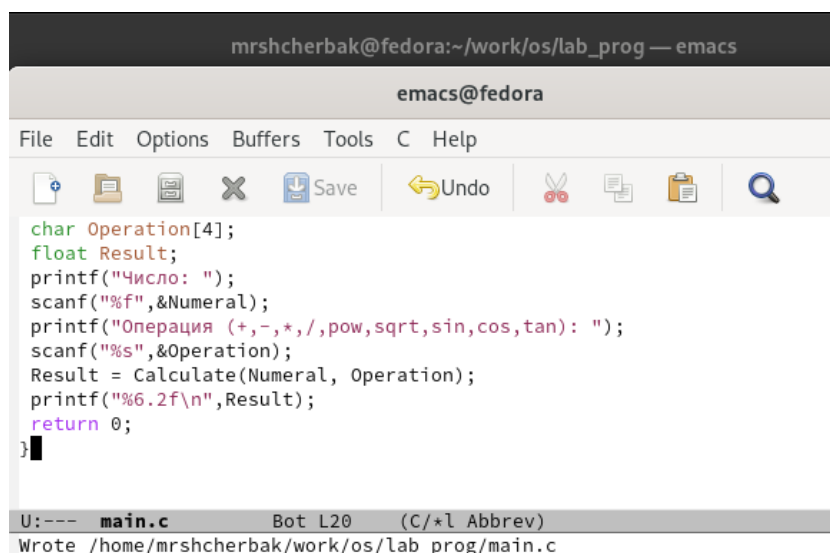
A screenshot of the Emacs editor window titled 'emacs@fedora'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The main text area shows the content of 'calculate.h' with the following code:

```
////////////////////////////////////  
// calculate.h  
  
#ifndef CALCULATE_H_  
#define CALCULATE_H_  
  
float Calculate(float Numeral, char Operation[4]);  
  
#endif /*CALCULATE_H_*/
```

The status bar at the bottom indicates 'U:--- calculate.h All L9 (C/*l Abbrev)'.

Рис. 3.4: Содержимое файла calculate.h

5. Аналогично открыла и написала основной файл main.c, реализующий интерфейс пользователя к калькулятору: (Рис. 3.5).

A screenshot of the Emacs editor window titled 'emacs@fedora'. The menu bar and toolbar are the same as in the previous image. The main text area shows the content of 'main.c' with the following code:

```
char Operation[4];  
float Result;  
printf("Число: ");  
scanf("%f",&Numeral);  
printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");  
scanf("%s",&Operation);  
Result = Calculate(Numeral, Operation);  
printf("%6.2f\n",Result);  
return 0;  
}
```

The status bar at the bottom indicates 'U:--- main.c Bot L20 (C/*l Abbrev)' and 'Wrote /home/mrshcherbak/work/os/lab_prog/main.c'.

Рис. 3.5: Содержимое файла main.c

6. Выполнила компиляцию программы посредством gcc: (Рис.3.6).

```
mrshcherbak@fedora:~/work/os/lab_prog
[mrshcherbak@fedora lab_prog]$ gcc -c calculate.c
[mrshcherbak@fedora lab_prog]$ gcc -c main.c
[mrshcherbak@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[mrshcherbak@fedora lab_prog]$
```

Рис. 3.6: Компиляция программы

7. Создала Makefile со следующим содержанием: (Рис.3.7 - Рис.3.8).

```
[mrshcherbak@fedora lab_prog]$ touch Makefile
[mrshcherbak@fedora lab_prog]$ ls
calcul calculate.c~ calculate.h~ main.c main.o
calculate.c calculate.h calculate.o main.c~ Makefile
[mrshcherbak@fedora lab_prog]$ emacs Makefile
```

Рис. 3.7: Создание Makefile

```
mrshcherbak@fedora:~/work/os/lab_prog — emacs Makefile
emacs@fedora
File Edit Options Buffers Tools Makefile Help
[Icons: New, Open, Save, Undo, Redo, Find, etc.]

#
# Makefile
#

CC = gcc
CFLAGS =
LIBS = -lm

calcul: calculate.o main.o
    gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    gcc -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
    gcc -c main.c $(CFLAGS)

clean:
    -rm calcul *.o *~

# End Makefile

U:*** Makefile All L14 (GNUmakefile)
```

Рис. 3.8: Содержимое Makefile

Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один испол-

няемый файл `calcul` (цель `calcul`). Цель `clean` нужна для автоматического удаления файлов. Переменная `CC` отвечает за утилиту для компиляции. Переменная `CFLAGS` отвечает за опции в данной утилите. Переменная `LIBS` отвечает за опции для объединения объектных файлов в один исполняемый файл.

8. Перед использованием `gdb` исправила `Makefile`: в переменную `CFLAGS` добавила опцию `-g`, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. (Рис.3.9).

```
#
# Makefile
#

CC = gcc
CFLAGS = -g
LIBS = -lm

calcul: calculate.o main.o
    gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    gcc -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
    gcc -c main.c $(CFLAGS)

clean:
    -rm calcul *.o *~

# End Makefile
```

Рис. 3.9: Исправленный `Makefile`

9. С помощью `gdb` выполнила отладку программы `calcul` (Рис.3.10).

- запустила отладчик GDB, загрузив в него программу для отладки
- Для запуска программы внутри отладчика ввела команду `run`:

```
[mrshcherbak@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 12.1-1.fc35
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./calcul...
(gdb) run
Starting program: /home/mrshcherbak/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 25
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): /
Делитель: 5
5.00
[Inferior 1 (process 13411) exited normally]
```

Рис. 3.10: Ввели число 25, затем операцию деления и делитель 5. Получили ответ 5

10. Использовала команду list: (Рис.3.11).

- Для постраничного просмотра исходного кода использовала команду list.
- Для просмотра строк с 12 по 15 основного файла использовала list с параметрами: list 12,15.
- Для просмотра определённых строк не основного файла использовала list с параметрами: list calculate.c:20,29

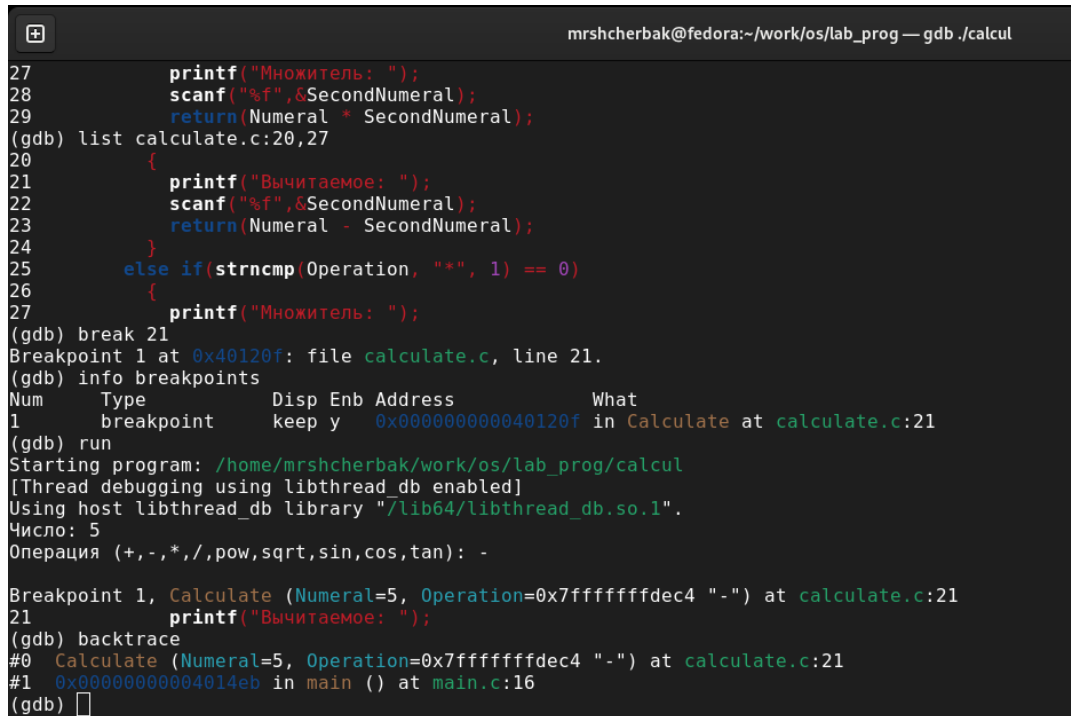
```
mrshcherbak@fedora:~/work/os/lab_prog — gdb ./calcul
Делитель: 5
5.00
[Inferior 1 (process 13411) exited normally]
(gdb) list
1  //////////////////////////////////////////////////
2  // main.c
3
4  #include <stdio.h>
5  #include "calculate.h"
6
7  int
8  main (void){
9      float Numeral;
10     char Operation[4];
(gdb) list
11     float Result;
12     printf("Число: ");
13     scanf("%f",&Numeral);
14     printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");
15     scanf("%s",&Operation);
16     Result = Calculate(Numeral, Operation);
17     printf("%b.2f\n",Result);
18     return 0;
19 }
(gdb) list 12,15
12     printf("Число: ");
13     scanf("%f",&Numeral);
14     printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");
15     scanf("%s",&Operation);
(gdb) list calculate.c:20,29
20 {
21     printf("Вычитаемое: ");
22     scanf("%f",&SecondNumeral);
23     return(Numeral - SecondNumeral);
24 }
25 else if(strncmp(Operation, "-", 1) == 0)
26 {
27     printf("Множитель: ");
28     scanf("%f",&SecondNumeral);
29     return(Numeral * SecondNumeral);
(gdb) █
```

Рис. 3.11: Команда list

11. Выполнение нижеперечисленных действий: (Рис.3.12).

- установила точку останова в файле calculate.c на строке номер 21:
list calculate.c:20,27
break 21
- вывела информацию об имеющихся в проекте точках останова: info
breakpoints
- запустила программу внутри отладчика и убедилась, что программа останавливается в момент прохождения точки останова:
run
5
“-” (просто минус)
backtrace
- отладчик выдал следующую информацию:
#0 Calculate (Numeral=5, Operation=0x7fffffffdec4 “-”) at calculate.c:21

#1 0x0000000000400b2b in main () at main.c:16



```
27     printf("Множитель: ");
28     scanf("%f",&SecondNumeral);
29     return(Numeral * SecondNumeral);
(gdb) list calculate.c:20,27
20     {
21         printf("Вычитаемое: ");
22         scanf("%f",&SecondNumeral);
23         return(Numeral - SecondNumeral);
24     }
25     else if(strncmp(Operation, "*", 1) == 0)
26     {
27         printf("Множитель: ");
(gdb) break 21
Breakpoint 1 at 0x40120f: file calculate.c, line 21.
(gdb) info breakpoints
Num  Type             Disp Enb Address            What
1      breakpoint      keep y   0x000000000040120f in Calculate at calculate.c:21
(gdb) run
Starting program: /home/mrshcherbak/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdec4 "-") at calculate.c:21
21     printf("Вычитаемое: ");
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdec4 "-") at calculate.c:21
#1 0x00000000004014eb in main () at main.c:16
(gdb) □
```

Рис. 3.12: Выполнение

12. выполнение нижеперечисленных действий (Рис.3.13).

Посмотрела, чему равно на этом этапе значение переменной Numeral, введя:

print Numeral

На экран должно быть выведено число 5.

Сравнила с результатом вывода на экран после использования команды:

display Numeral

Убрала точки останова:

info breakpoints

delete 1

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num      Type      Disp Enb Address      What
1        breakpoint keep y 0x000000000040120f in calculate at calculate.c:21
breakpoint already hit 1 time
(gdb) delete 1
```

Рис. 3.13: Выполнение

13. С помощью утилиты splint попробовала проанализировать коды файлов calculate.c и main.c. (Рис.3.14 - Рис.3.15).

```
mrshcherbak@fedora:~/work/os/lab_prog
[mrshcherbak@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: Function parameter Operation declared as manifest array (size
constant is meaningless)
A formal parameter is declared as an array with size. The size of the array
is ignored in this context, since the array formal parameter is treated as a
pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:10:31: Function parameter Operation declared as manifest array
(size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:7: Return value (type int) ignored: scanf("%f", &Sec...
Result returned by function call is not used. If this is intended, can cast
result to (void) to eliminate message. (Use -retvalint to inhibit warning)
calculate.c:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: Dangerous equality comparison involving float types:
SecondNumeral == 0
Two real (float, double, or long double) values are compared directly using
== or != primitive. This may produce unexpected results since floating point
representations are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:38:13: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:46:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:13: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:50:11: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:52:11: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:54:11: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:56:11: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:60:13: Return value type double does not match declared type float:
(HUGE_VAL)
Finished checking --- 15 code warnings
```

Рис. 3.14: Splint calculate.c

```

[mrshcherbak@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: Function parameter Operation declared as manifest array (size
        constant is meaningless)
    A formal parameter is declared as an array with size. The size of the array
    is ignored in this context, since the array formal parameter is treated as a
    pointer. (Use -fixedformalarray to inhibit warning)
main.c: (in function main)
main.c:13:2: Return value (type int) ignored: scanf("%f", &Num...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
main.c:15:13: Format argument 1 to scanf (%s) expects char * gets char [4] *:
    &Operation
    Type of parameter is not consistent with corresponding code in format string.
    (Use -formattype to inhibit warning)
    main.c:15:10: Corresponding format code
main.c:15:2: Return value (type int) ignored: scanf("%s", &ope...

Finished checking --- 4 code warnings
[mrshcherbak@fedora lab_prog]$

```

Рис. 3.15: Splint main.c

С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохраняются. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потере данных.

4 Контрольные вопросы:

1). Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой тап или опцией -help(-h) для каждой команды.

2). Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения; проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования; непосредственная разработка приложения: кодирование – по сути создание исходного текста программы (возможно в нескольких вариантах); –анализ разработанного кода; сборка, компиляция и разработка исполняемого модуля; отестирование и отладка, сохранение произведённых изменений; документирование. Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др. После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

3). Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .c воспринимаются gcc как программы на языке C, файлы с расширением .cpp или .C – как файлы на языке C++, а файлы с расширением .o считаются объектными. Например, в команде «gcc -o main.c»: gcc по расширению (суффиксу) .c распознает тип

файла для компиляции и формирует объектный модуль – файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c». В ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

4). Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.

5). Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.

6). Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис: ... : ... <команда 1> ... Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции. В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды – собственно действия, которые необходимо выполнить для достижения цели. Общий синтаксис Makefile имеет вид: target1 [target2...]: [dependment1...][(tab) commands] [#commentary][(tab) commands] [#commentary]. Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться). Одинарное двоеточие указывает на то, что последовательность команд должна содержаться в одной строке. Для переноса можно в длинной строке команд

можно использовать обратный слэш (`\`). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках. Пример более сложного синтаксиса Makefile: `## Makefile for abcd.c`
`CC = gcc`
`CFLAGS =`
`# Compile abcd.c normally`
`abcd: abcd.c(CFLAGS) abcd.o`
`clean: -rm abcd.o`
`~# End Makefile for abcd.c`. В этом примере в начале файла заданы три переменные: `CC` и `CFLAGS`. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем `clean` производит очистку каталога от файлов, полученных в результате компиляции. Для её описания использованы регулярные выражения.

7). Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией `-g` компилятора `gcc`: `gcc -g file.c`. После этого для начала работы с `gdb` необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: `gdb file.o`

8). Основные команды отладчика `gdb`:

- `backtrace` – вывод на экран пути к текущей точке останова (по сути вывод – названий всех функций);
- `break` – установить точку останова (в качестве параметра может быть указан номер строки или название функции);
- `clear` – удалить все точки останова в функции;
- `continue` – продолжить выполнение программы;
- `delete` – удалить точку останова;
- `display` – добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы;

`finish` – выполнить программу до момента выхода из функции;

`info breakpoints` – вывести на экран список используемых точек останова;

`info watchpoints` – вывести на экран список используемых контрольных выражений;

`list` – вывести на экран исходный код (в ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями. качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк);

`next` – выполнить программу пошагово, но без выполнения вызываемых в программе функций;

`print` – вывести значение указываемого в качестве параметра выражения;

`run` – запуск программы на выполнение;

`set` – установить новое значение переменной;

`step` – пошаговое выполнение программы;

`watch` – установить контрольное выражение, при изменении значения которого программа будет остановлена. Для выхода из `gdb` можно воспользоваться командой `quit` (или её сокращённым вариантом `q`) или комбинацией клавиш `Ctrl-d`. Более подробную информацию по работе с `gdb` можно получить с помощью команд `gdb-hi` и `mangdb`.

9). Схема отладки программы показана в 6 пункте лабораторной работы.

10). При первом запуске компилятор не выдал никаких ошибок, но в коде программы `main.c` допущена ошибка, которую компилятор мог пропустить (возможно, из-за версии 8.3.0-19): в строке `scanf("%s", &Operation);` нужно убрать знак `&`, потому что имя массива символов уже является указателем на первый элемент этого массива.

11). Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся: `cscope` – исследование

функций, содержащихся в программе, lint – критическая проверка программ, написанных на языке Си.

12). Утилита splint анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки. В отличие от компилятора Санализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.

5 Выводы

Таким образом, в ходе ЛРН^{№13} я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.